Introduction to Haskell IV

Rolf Fagerberg

Spring 2005

Algebraic Types

Beside the simple type synonymes (using the keyword `type`), more advanced user defined types - denoted algebraic types - can be created with the `data` keyword.

The general syntax is:

```
data Typename zero_or_more_type-variables
  = Constructor1 zero_or_more_types |
  = Constructor2 zero_or_more_types |
  ⋮
  = Constructor3 zero_or_more_types
  deriving (list_of_certain_classes)
```

The identifiers for the type name and the constructor names must be capitalized.

Examples

Enumerated types:

```
data Bool = False | True
data Ordering = LT | EQ | GT
data Seasons = Winter | Spring | Summer | Fall
data WeekDays
  = Mon | Tue | Wed | Thu | Fri | Sat | Sun

workDays = [Mon, Tue, Wed, Thu, Fri]
```

Product types (alias tuples, alias records):

```
data DBRecord = DBRec Name Address Age
type Name = String
type Address = String
type Age = Int

person1 = DBRec "Joe Dole" "Main Street 10" 42
```

Examples

Alternatives:

```
data Shape
  = Circle Float | Rectangle Float Float
```

Note: constructors are functions:

```
Circle ::  Float -> Shape
shape1 = Circle 3.0
Rectangle ::  Float -> Float -> Shape
shape2 = Rectangle 45.9 87.6
```

Additionally, they can (like the built-in constructors `[]`, `:`, etc.) be used as patterns in pattern matching:

```
area ::  Shape -> Float
area (Circle r)      = pi*r*r
area (Rectangle w h) = w*h
```

Examples

Algebraic types can be recursive:

```
data IntList = EmptyList | Cons Int IntList

data IntExpr = Literal Int |
               Add IntExpr IntExpr |
               Sub IntExpr IntExpr

data IntTree = IntLeaf |
               IntNode Int IntTree IntTree

tree = IntNode 7 IntLeaf (IntNode 13 IntLeaf IntLeaf)
```

Constructors can be infix operators (identifier must then start with ´:´):

```
data IntList = EmptyList | Int ::: IntList
```

Examples

Algebraic types can be parametric:

```
data List a = EmptyList | Cons a (List a)

data Tree a = Leaf |
              Node a (Tree a) (Tree a)
```

Example functions on trees:

```
depth :: Tree a -> Int
depth Leaf         = 0
depth (Node _ l r) = 1 + max (depth l) (depth r)

inorder :: Tree a -> [a]
inorder Leaf         = []
inorder (Node x l r) = inorder l ++ [x] ++ inorder r
```

Deriving Membership of Classes

Membership of certain standard type classes can be generated automatically in Haskell:

```
data WeekDays
  = Mon | Tue | Wed | Thu | Fri | Sat | Sun
  deriving (Eq, Ord, Enum, Show, Read)
```

The operations of the classes are automatically defined using obvious (recursive) definitions (with ordering going from left to right, and using analogy with lexicographic ordering for recursive structures). The derivation of `Enum` can only be done for enumeration types (nullary constructors only).

```
[Mon,Wed .. Sat]  ⇝  [Mon,Wed,Fri]
```